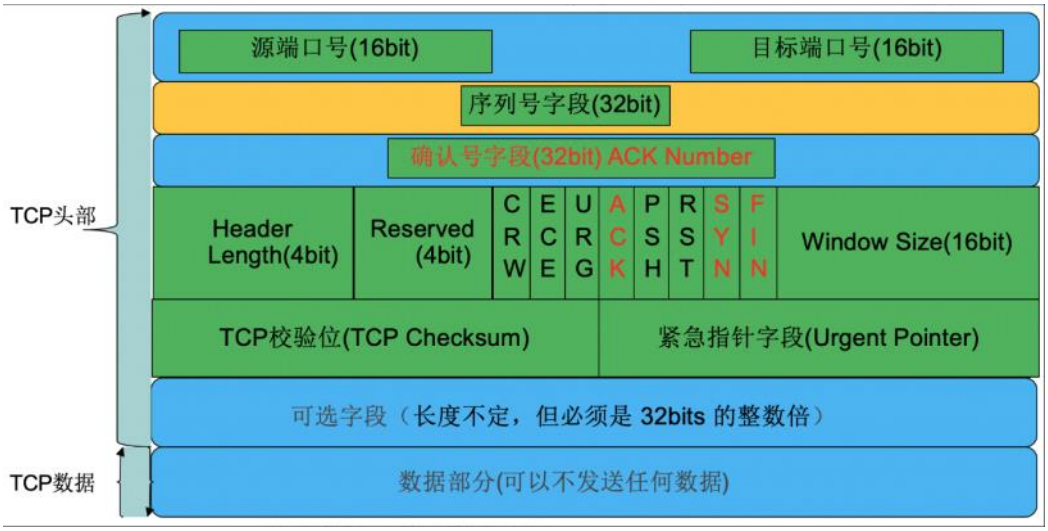
TCP协议

2024年6月27日　　16:12

三、TCP协议

3.1 TCP协议首部

传输层（TCP、UDP协议）



TPC的首部，至少是20个字节

TCP的协议栈会将传入的参数，放到TCP首部中。

端口号（16比特位）：因此端口号的范围是0-65536

Header Length（首部长度）：4比特位，0-15，单位是4个字节，也就是说，Header Length的值，最小是5（5*4，20个字节），因此首部的长度最大为60个字节，可选字段最大为40个字节。

序列号字段：确定数据流的顺序

确认号字段：接收方确定接收成功

一个比特位的ACK为0，代表确认字号段无效，为1，确认号字段有效

SYN为1，代表该数据包是握手请求的数据包，为0，代表正常通信数据包

TCP协议的特性：
TCP协议是一个可靠的传输层协议
TCP如何保证可靠性？
　　面向连接 ：**建立连接需要三次握手（三个数据包的交互，在通信双方的内存中记录了本次连接的情况）**

UDP协议是一个不可靠的传输层协议
UDP无连接

3.2 TCP连接的建立与终止

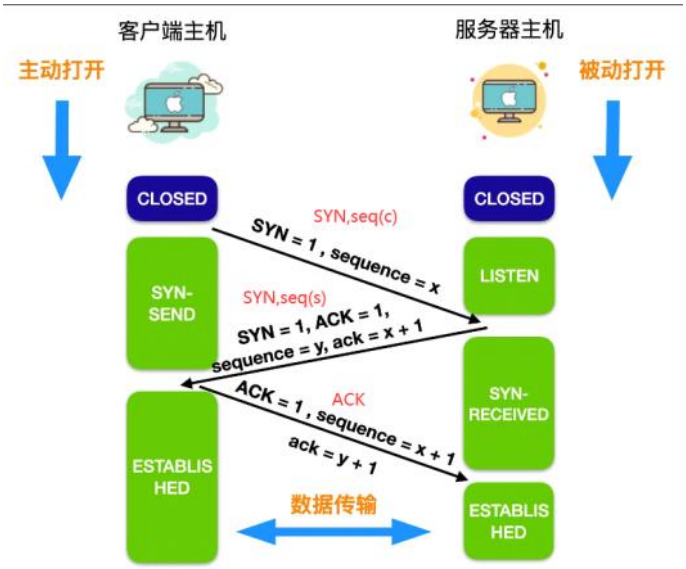
TCP 连接指的是在两台计算机进行通信之前，先通过三个数据包的交互建立连接，被称为TCP 三次握手，而不是不管三七二十一上来接直接发送，也不管对方现在方不方便接收。这就像是我们在和别人打电话的时候说话交流之前先相互喂喂喂几次，在确定了相互能听到对方的声音时才进入主题类似。这里的连接并不是拿两根导线把你们通信的双方连上，而是通过三个数据包的交互在连接的两个端上分别使用相应的数据结构记录当前存在的连接情况。

在这两个端都无数据发送时需要通过四个数据包断开连接。（四次挥手）

在发送数据前，相互通信的双方(即发送方和接受方)需要建立一条连接，在发送数据后，通信双方需要断开连接，这就是TCP连接的建立和终止。

TCP的连接建立->终止总共分为三个阶段

3.1.1 TCP建立连接-三次握手



seq：序列号，用来确保数据的顺序

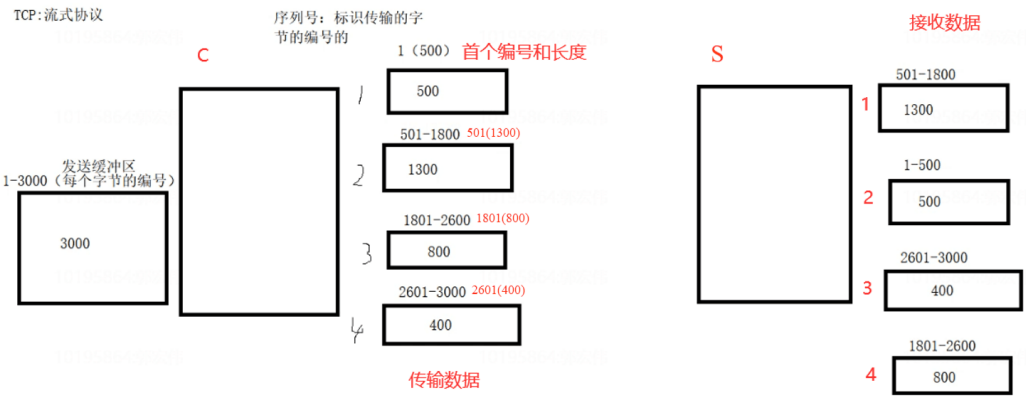
seq(c):初始序列号：数据流的编号，不一定是从1开始的（是随机开始的），具体从哪一个值开始，客户端要发送给服务器。不然无法确定这个数据流前面是否还存在数据。

seq(s):服务器的初始序列号。

序列号的作用

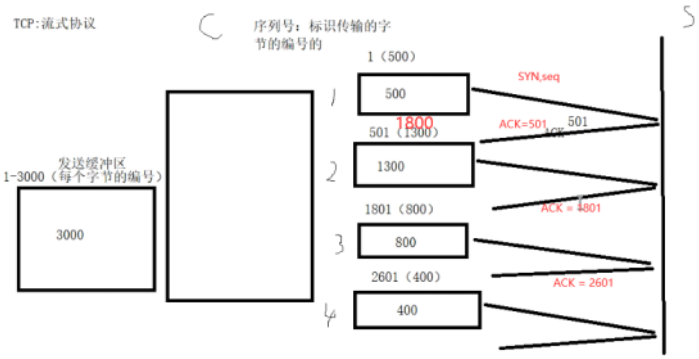
TCP协议是流式协议：这样就决定了在在传输层，TCP协议会将数据按照流的方式进行处理（可以在任意位置进行分割）数据会被任意分割为不同大小的数据进行发送，而接收方收到数据顺序不一定是刚好按照顺序接收到的，因此我们需要一个序列号来标识传输字节的编号。

但是，我们不可能传输数据流中的每一个字节的编号，只需要发送**一个编号加一个长度**即可。这样，接收方拿到数据之后，就能知道该如何组合数据了



ACK：确认序号，用来确保接收方收到数据

上面讲了，传输层如何将数据流发送给接收方，那发送方如何确定接收方接收到了数据呢？
接收方收到数据包之后，会向发送方返回一个数据包，里面包含一个ACK确认号，来通知发送方，收到数据
ack= x+1



一个比特位的ACK为0，代表确认字号段无效

SYN为1，代表是握手请求的数据包，为0，代表正常通信数据包

三次握手的流程为

1、客户端向服务器发送一个数据包（SYN=1,seq(c)=x），请求建立连接
2、服务器接收到客户端请求建立连接的数据包后，如果同意建立连接，就向客户端回复一个同意连接的数据包（SYN=1,seq(s)=y,ACK=1，ack = x+1）
3、客户端接收到服务器同意连接的请求后，回复服务器一个确认连接的数据包（ACK=1，seq(c)=x+1,ack=y+1）

三次握手详细流程

1、服务端进程准备好接收来自外部的TCP连接，一般情况下是调用socket、bind、listen三个函数完成。
这种打开方式被认为是被动打开(passiveopen)。
然后服务端进程处于LISTEN状态，等待客户端连接请求。

2、客户端通过connect发起主动打开(active open)，向服务器发出连接请求，
请求中首部同步位SYN=1，同时选择一个初始序号sequence，简写seg=x。
SYN 报文段不允许携带数据，只消耗一个序号。此时，客户端进入SYN-SEND 状态。

3、服务器收到客户端连接后，需要确认客户端的报文段。
在确认报文段中，把SYN和ACK位都置为1。确认号是ack=x+1，同时也为自己选择一个初始序号seq=y。
这个报文段也不能携带数据，但同样要消耗掉一个序号。
此时，TCP服务器进入SYN-RECEIVED(同步收到)状态。

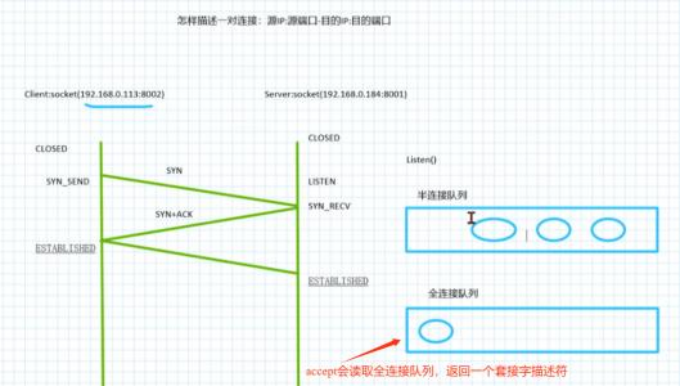
4、客户端在收到服务器发出的响应后，还需要给出确认连接。
确认连接中的ACK置为1，序号为seq=x+1，确认号为ack=y+1。
TCP 规定，这个报文段可以携带数据也可以不携带数据，
如果不携带数据，那么下一个数据报文段的序号仍是seq=x+1。
这时，客户端进入ESTABLISHED(已连接)状态。

5、服务器收到客户的确认后，也进入ESTABLISHED状态。

这是一个典型的三次握手过程，通过上面3个报文段就能够完成一个TCP连接的建立。三次握手的的目的不仅仅在于让通信双方知晓正在建立一个连接，也在于利用数据包中的选项字段来交换一些特殊信息，交换初始序列号。一般首个发送SYN 报文的一方被认为是主动打开一个连接，而这一方通常也被称为客户端。而SYN的接收方通常被称为服务端，它用于接收这个SYN，并发送下面的SYN，因此这种打开方式是被动打开。

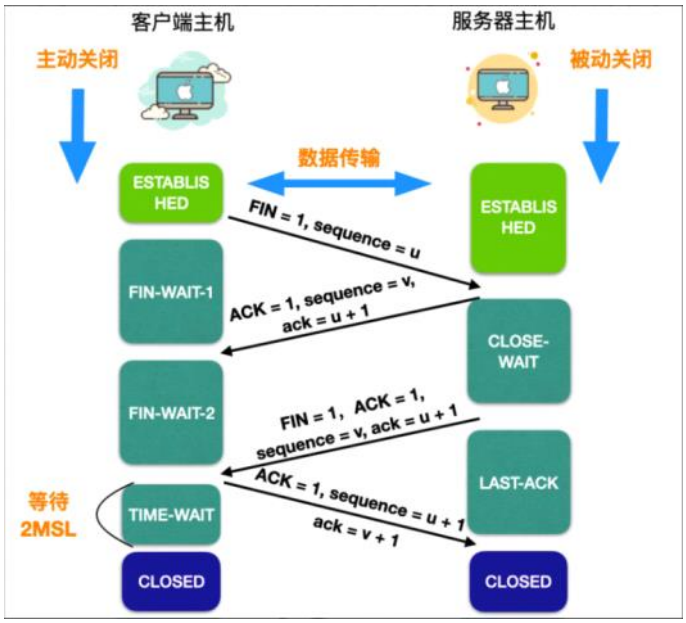
三次握手的最后一次握手是可以携带数据的

TCP 建立一个连接需要三个报文段，释放一个连接却需要四个报文段。



3.1.2 TCP断开连接-四次挥手

数据传输结束后，通信的双方可以释放连接。数据传输结束后的客户端主机和服务端主机都处于ESTABLISHED状态，然后进入释放连接的过程。



TCP断开连接需要历经的过程如下

1、客户端应用程序发出释放连接的报文段，并停止发送数据，主动关闭TCP连接。客户端主机发送释放连接的报文段，报文段中首部FIN位置为1，不包含数据，序列号位seq=u，此时客户端主机进入FIN-WAIT-1(终止等待1)阶段。

2、服务器主机接受到客户端发出的报文段后，即发出确认应报文，确认应答报文中ACK=1，生成自己的序号位

seg=v，ack=u+1，然后服务器主机就进入 CLOSE-WAIT(关闭等待)状态:

这里CLOSE-WAIT状态的作用是，为了防止服务器还有未发送完的数据包，此时服务器是可以向客户端发送数据包的。这个阶段，服务器要尽快将剩余的数据包完成处理，一旦服务器将剩余的内容，就会进入下一状态。

3、客户端主机收到服务端主机的确认应答后，即进入FIN-WAIT-2(终止等待2)的状态。等待客户端发出连接释放的报文段。

4、这时服务端主机会发出断开连接的报文段，报文段中ACK=1，序列号seq=v，ack=u+1，在发送完断开请求的报文后，服务端主机就进入了LAST-ACK(最后确认)的阶段。

5、客户端收到服务端的断开连接请求后，客户端需要作出响应，客户端发出断开连接的报文段，在报文段中，ACK=1，序列号seq=u+1，因为客户端从连接开始断开后就没有再发送数据，ack=v+1，然后进入到TIME-WAIT(时间等待)状态，请注意，这个时候TCP连接还没有释放。必须经过时间等待的设置，也就是2MSL后，客户端才会进入CLOSED状态，时间MSL叫做最长报文段寿命(Maximum Segment Lifetime)。

这里的作用是防止，客户端发回的ACK确认报文丢失，因为客户端发送的最后一个ACK报文是没有重传机制的，而服务器在LAST-ACK之前发送的出断开连接的报文段，在未收到回复时，就会进行重传，客户端在TIME-WAIT阶段再次接收到服务器发送的出断开连接的报文段，就知道之前发的包丢了，就会再发一个ACK确认报。

6、服务端主要收到了客户端的断开连接确认后，就会进入CLOSED状态。因为服务端结束TCP连接时间要比客户端早，而整个连接断开过程需要发送四个报文段，因此释放连接的过程也被称为四次挥手。

3.1.3 TCP半开启

TCP连接处于半开启的这种状态是因为连接的一方关闭或者终止了这个TCP连接却没有通知另一方，也就是说两个人正在微信聊天，你下线了你不告诉我，我还在跟你侃八卦呢。此时就认为这条连接处于半开启状态。这种情况可能发生在通信中的一方处于主机崩溃的情况下，你xxx的，我电脑死机了我咋告诉你?

只要处于半开启状态的一方不传输数据的话，那么是无法检测出来对方主机已经下线的。另外一种处于半开启状态的原因是通信的一方关闭了主机电源而不是正常关机。这种情况下会导致服务器上有很多半开启的TCP连接。

解决方法:

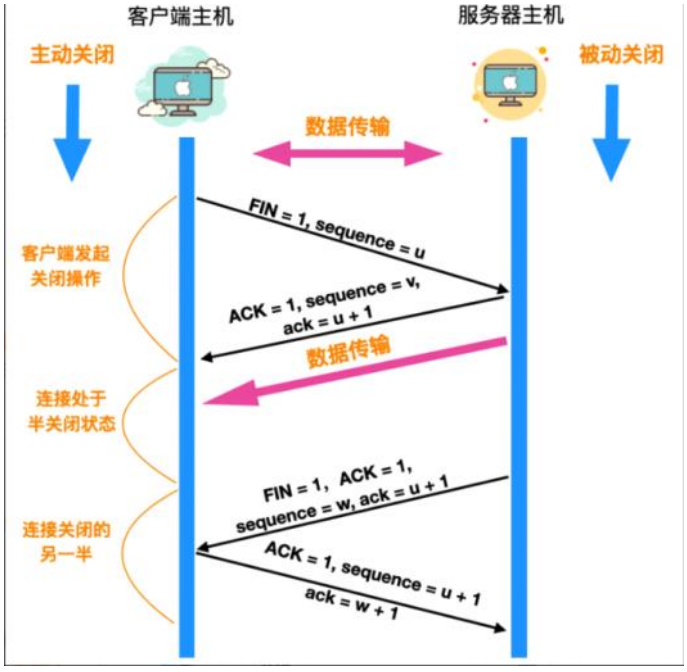
如何解决半开启问题呢，引入**心跳机制**就可以察觉半开启。（每隔一段时间就发一个数据包检测一下连接是否还在）

如果需要发数据的话，死机的这边收到之后发现这个连接并不存在了，就会回复RST包告知，这个时候就需要重新建立连接了。

3.1.4 TCP半关闭

就是客户端像服务器发送FIN请求，客户端回复ACK，客户端接收到了服务器的确认请求的ACK，之后，客户端就不能再向服务器发送数据了，只能由服务器向客户端发送数据，此时客户端的状态就是半关闭状态。

既然 TCP支持半开启操作，那么我们可以设想TCP也支持半关闭操作。同样的，TCP半关闭也并不常见。TCP的半关闭操作是指仅仅关闭数据流的一个传输方向。两个半关闭操作合在一起就能够关闭整个连接。在一般情况下，通信双方会通过应用程序互相发送FIN报文段来结连接，但是在TCP半关闭的情况下，应用程序会表明自己的想法:"我已经完成了数据的发送，并发送了一个FIN报文段给对方，但是我依然希望接收来自对方的数据直到它发送一个FIN报文段给我"。下面是一个TCP 半关闭的示意图。



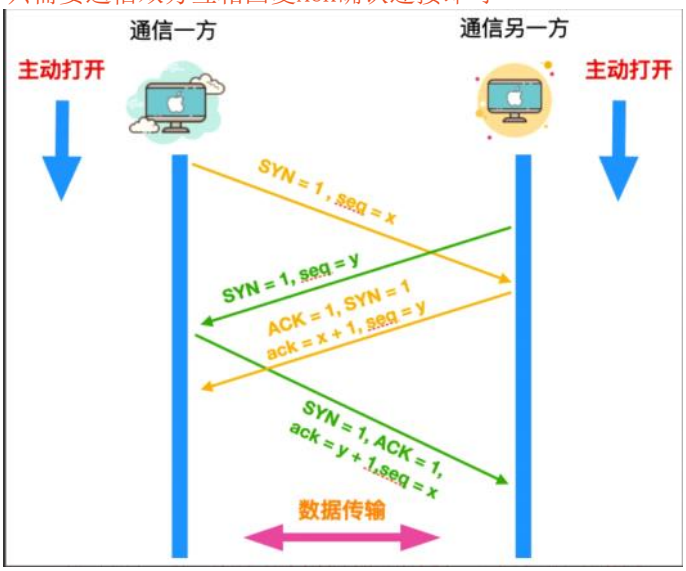
解释一下这个过程:

首先客户端主机和服务器主机一直在进行数据传输，一段时间后，客户端发起了FIN 报文，要求主动断开连接，服务器收到FIN后，回应ACK，由于此时发起半关闭的一方也就是客户端仍然希望服务器发送数据，所以服务器会继续发送数据，一段时间后服务器发送另外一条FIN报文，在客户端收到FIN报文回应ACK给服务器后，断开连接。TCP的半关闭操作中，连接的一个方向被关闭，而另一个方向仍在传输数据直到它被关闭为止。只不过很少有应用程序使用这一特性。

3.1.5 同时打开、同时关闭

1、同时打开---通信双方同时申请建立连接

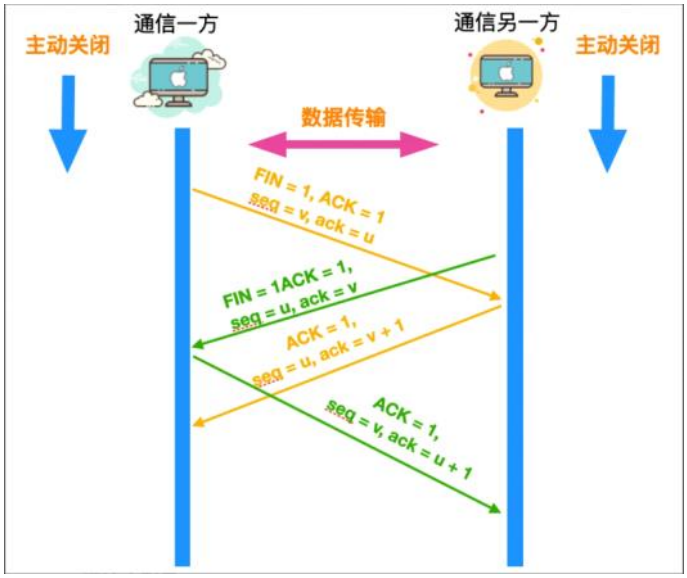
此时只需要通信双方互相回复ACK确认连接即可



如上图所示，通信双方都在收到对方报文前主动发送了 SYN 报文，都在收到彼此的报文后回复了一个 ACK 报文。一个同时打开过程需要交换四个报文段，比普通的三次握手增加了一个，由于同时打开没有客户端和服务器一说，所以这里我用了通信双方来称呼。

2、同时关闭---通信双方同时申请断开连接

像同时打开一样，同时关闭也是通信双方同时提出主动关闭请求，发送FIN报文，下图显示了一个同时关闭的过程。



同时关闭过程中需要交换和正常关闭相同数量的报文段，只不过同时关闭不像四次挥手那样顺序进行，而是交叉进行的。

3.1.6 初识序列号

初始序列号它是有专业术语表示的，初始序列号的英文名称是Initialsequencenumbers(ISN)，所以我们上面表示的seq=v，其实就表示的ISN。在发送SYN 之前，通信双方会选择一个初始序列号。初始序列号是随机生成的，每一个TCP连接都会有一个不同的初始序列号。RFC文档指出初始序列号是一个32位的计数器，每4us(微秒)+1。因为每个TCP连接都是一个不同的实例，这么安排的目的就是为了防止出现序列号重叠的情况。

初始序列号是一个32位的计数器，每4us(微秒)+1。（现在还要加上源IP、源端口：目的IP、目的端口的哈希值）

当一个 TCP连接建立的过程中，只有正确的TCP四元组和正确的序列号才会被对方接收。这也反应了TCP报文段容易被伪造的脆弱性，因为只要我伪造了一个相同的四元组和初始序列号就能够伪造 TCP 连接，从而打断 TCP 的正常连接，所以抵御这种攻击的一种方式就是使用初始序列号，另外一种方法就是加密序列号。

3.2 TCP状态转换

3.2.5 TIME_WAIT状态

通信双方建立TCP连接后，主动关闭连接的一方就会进入TIME_WAIT状态。TIME WAIT状态也称为2MSL的等待状态。在这个状态下，TCP将会等待最大段生存期(Maximum Segment Lifetime,MSL)时间的两倍。

这里需要解释下MSL

MSL是TCP段期望的最大生存时间，也就是在网络中存在的最长时间。这个时间是有限制的，因为我们知道TCP是依靠IP数据段来进行传输的，IP数据报中有TTL字段，这个字段决定了IP报文 的生存时间，一般情况下，**TCP 的最大生存时间是2分钟**，不过这个数值是可以修改的，根据不同操作系统可以修改此值。

基于此，我们来探讨TIMEWAIT的状态。当TCP执行一个主动关闭并发送最终的ACK时，TIMEWAIT应该以2*最大生存时间存在，这样就能够让 TCP 重新发送最终的 ACK 以避免出现丢 失的情况。重新发送最终的 ACK 并不是因为 TCP 重传了 ACK，而是因为通信另一方重传了 FIN，客户端经常会发送FIN，因为它需要ACK的响应才能够关闭连接，如果生存时间超过了2MSL的话，客户端就会发送RST，使服务端出错。

另一个影响2MSL等待状态的因素是当TCP处于等待状态时,通信双方将该连接(客户端 IP地址、客户端端口号、服务器IP地址、服务器端口号)定义为不可重新使用。只有当2MSL等待结束时,或一条新连接使用的初始序列号超过了连接之前的实例

所使用的最高序列号时,或者允许使用时间戳选项来区分之前连接实例的报文段以避免混淆时,这条连接才能被再次使用。不幸的是,一些实现施加了更加严格的约束。在这些系统中,如果一个端口号被处于 2MSL 等待状态的任何通信端所用,那么该端口号将不能被再次使用。

每个数据包都会带一个时间戳选项，收到一个数据包的时间戳比这个连接三次握手的时间都早，那么这个数据包就肯定不属于这个连接了。

1、保证四次挥手的可靠完成。
2、防止旧的重复的报文影响新的连接（有个包在断开连接之前阻塞在网络中了，在断开连接之后还没发送到，然后双方又建立了一个新的连接，此时之前连接阻塞的包发送到了，这样就会出现问题）

IP协议中的TTL字段（time to live，单位是跳）

是指数据包在网络层中允许转发的最大次数，8位，0-255，初始值一般是64，数据包在网络路由器中，每经过一个路由器转发，TTL就会减一，直到TTL=0，那么网络层就认为，接收这个 数据包的路由器出问题了，该数据包就会被销毁。不然数据包就会占满整个网络层。那也就是说每个数据包在网络中都会有一个存活时间，过了这个时间就会消失。

3.3 TCP超时和重传

通信双方，某一方发送一个数据包后，内部会自己维护一个计时器，一旦在规定是时间内，还未收到另一方确认收到的ACK，发送消息的一方就有理由认为这个数据包丢了（有两种可能：数据包没发送到，确认回复的ACK丢失,这种ACK丢失的情况称为伪超时），此时发送方就会再次发送这个数据包，这种操作就称为超时和重传。

但是，如果当前网络很差，发送方每次的重传都没有收到ACK回复，就会停止发送，系统内维护一个字段RTO（即超时等待时间），每一次重传RTO都会翻倍，也就是会多等待一倍时间，达到某个值之后，就会发送RST，断开连接。

没有永远不出错误的通信，这句话表明着不管外部条件多么完备，永远都会有出错的可能。所以，在 TCP 的正常通信过程中，也会出现错误，这种错误可能是由于数据包丢失引起的，也可能是由于数据包重复引起的，甚至可能是由于数据包失序引起的。

TCP的通信过程中，会由TCP的接收端返回一系列的确认信息来判断是否出现错误，一旦出现丢包等情况，TCP 就会启动重传操作，重传尚未确认的数据。

TCP 的重传有两种方式,种是基于时间，一种是基于确认信息，一般通过确认信息要比通过时间更加高效。

所以从这点就可以看出，TCP的确认和重传，都是基于数据包是否被确认为前提的。

TCP 在发送数据时会设置一个定时器，如果在定时器指定的时间内未收到确认信息，那么就会触发相应的超时或者基于计时器的重传操作，计时器超时通常被称为重传超时(RTO)

但是有另外一种不会引起延迟的方式，这就是快速重传

TCP 在每次超时重传一次报文后，其重传时间都会加倍，这种"间隔时间加倍"被称为二进制指数补偿(binary exponential backoff)。等到间隔时间加倍到 15.5min后，客户端会显示。

```
Connection closed by foreign host.
```

TCP拥有两个值来决定如何重传一个报文段，这两个值被定义在RFC[RCF1122]中，**第一个值是 R1，它表示愿意尝试重传的次数，第二个值 R2 表示TCP应该放弃连接的时间。**R1和R2应至少设为三次重传和100秒放弃TCP连接。

这里需要注意下，对连接建立报文SYN来说，它的R2至少应该设置为3分钟，但是在不同的系统中，R1和R2值的设置方式也不同。

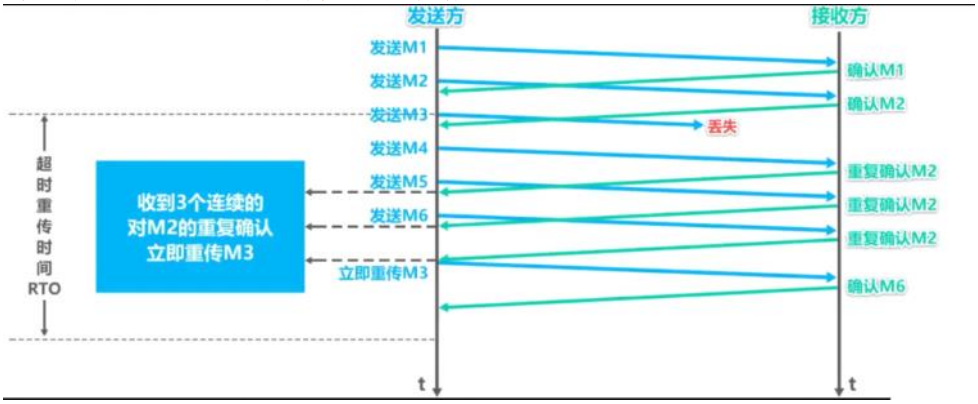
在Linux系统中，R1和R2的值可以通过应用程序来设置，或者是修改net.ipv4.tcp_retriesl和net.ipv4.tcp_retries2的值来设置。变量值就是重传次数。

tcp_retries2的默认值是 15，这个重传次数的耗时大约是13-30分钟，这只是一个大概值，最终耗时时间还要取决于RTO，也就是重传超时时间。tcpretries1的默认值是3。

对于 SYN 段来说,net.ipv4.tcp_syn_retries 和net.ipv4.tcp_synack_retries这两个值限制了SYN的重传次数，默认是5，大约是180秒。

3.4 TCP快速重传

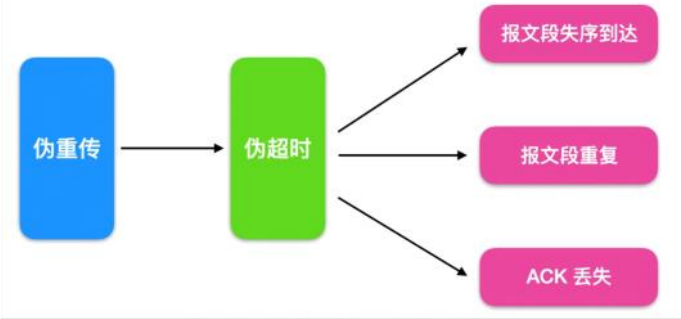
当连续收到三个冗余的ACK，TCP立马重传丢失的数据包



超时重传大概率是因为网络拥塞导致的，而快重传就只是单纯因为丢包导致的（而不是网络拥塞导致的丢包）

3.6 伪超时和重传

在某些情况下，即使没有出现报文段的丢失也可能会引发报文重传。这种重传行为被称为伪重传(spurious retransmission)，这种重传是没有必要的，造成这种情况的因素可能是由于伪超时(spurious timeout)，伪超时的意思就是过早的判定超时发生。造成伪超时的因素有很多，比如报文段失序到达，报文段重复，ACK丢失等情况。



检测和处理伪超时的方法有很多，这些方法统称为检测算法和响应算法。检测算法用于判断是否出现了超时现象或出现了计时器的重传现象。一旦出现了超时或者重传的情况，就会执行响应算法撤销或者减轻超时带来的影响，下面是几种算法，暂不深入这些实现细节。

为处理伪超时问题提出了许多方法。这些方法通常包含检测(detection)算法与响应(response)算法。检测算法用于判断某个超时或基于计时器的重传是否真实,一旦认定出现伪超时则执行响应算法,用于撤销或减轻该超时带来的影响。我们只讨论伪超时和伪重传的行为，不讨论具体的检测和处理算法。

3.8 TCP的流量控制和窗口管理

应用题，一个水池，上面灌水，下面出水，如果灌水比出生快，水池很快就会满了，然后溢出。

相比于通信的双方，一个发送消息，一个接收消息，一般发送消息的一方是速度较快的，可以一直发，而接收消息的一方，需要对数据进行处理，接收就比较慢，来不及处理的消息就会放在网关的缓冲区，一旦发送的消息很多，缓冲区也会满，就像水池一样溢出，就会造成数据丢失。

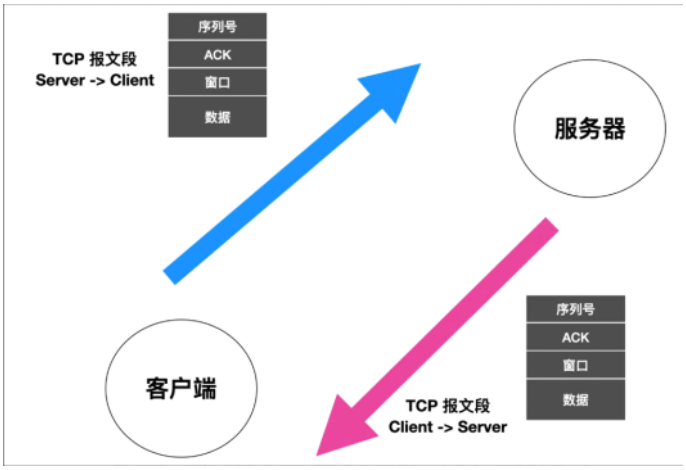
如果我们对发送方进行控制的话，就会造成上面的结果，因此要进行TCP的流量控制。

TCP采用头部的字段 窗口大小来让发送方明白，接收方的缓冲区大小，以此来让发送方确定，是否能够继续发送消息，在三次握手第二次接收方发送确认的ACK时，会带一个接收方自己的窗口大小，每一次的确认ACK，都会将这个窗口大小的字段进行更新。

确定发送方发送数据的速度与大小，以及接收方能够接收数据的大小，称之为TCP的流量控制

而TCP的流量控制是由滑动窗口管理实现的，又称**滑动窗口算法**

我们前面提过可以使用滑动窗口来实现流量控制，也就是说，客户端和服务器可以相互提供数据流信息的交换，数据流的相关信息主要包括**报文段序列号、ACK号和窗口大小。**

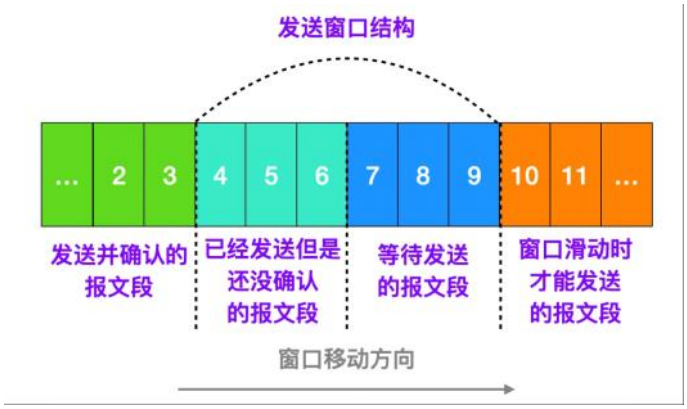


图中的两个箭头表示数据流方向，数据流方向也就是TCP报文段的传输方向。可以看到，每个TCP报文段中都包括了序号、ACK和窗口信息，可能还会有用户数据。TCP 报文段中的窗口大小表示接收端还能够接收的缓存空间的大小，以字节为单位。这个窗口大小是一种动态的，因为无时无刻都会有报文段的接收和消失，这种动态调整的窗口大小我们称之为滑动窗口，下面我们就来具体认识一下滑动窗口。

3.8.1 滑动窗口

TCP连接的每一端都可以发送数据，但是数据的发送不是没有限制的，实际上，TCP连接的两端都各自维护了一个发送窗口结构(send windowstructure)和接收窗口结构(receive window structure)，这两个窗口结构就是数据发送的限制。

3.8.2 发送方窗口



在这幅图中，涉及滑动窗口的四种概念:

- **已经发送并确认的报文段:**发送给接收方后，接收方回复ACK来对报文段进行响应，图中标注绿色的报文段就是已经经过接收方确认的报文段。
- **已经发送但是还没确认的报文段:**图中绿色区域是经过接收方确认的报文段，而浅蓝色这段区域指的是已经发送但是还未经过接收方确认的报文段。
- **等待发送的报文段:**图中深蓝色区域是等待发送的报文段，它属于发送窗口结构的一部分，也就是说，发送窗口结构其实是由已发送未确认+等待发送的报文段构成。
- **窗口滑动时才能发送的报文段:**如果图中的[4,9]这个集合内的报文段发送完毕后，整个滑动窗口会向右移动，图中橙色区域就是窗口右移时才能发送的报文段。

滑动窗口也是有边界的，这个边界是Leftedge和Rightedge，Left edge 是窗口的左边界，Rightedge是窗口的右边界。

当Left edge 向右移动而 Right edge 不变时，这个窗口可能处于 close 关闭状态。随着已发送的数据逐渐被确认从而导致窗口变小时，就会发生这种情况。

3.8.5.延时确认

TCP并不是每个数据包都会回复确认的ACK，多数情况下是 2 3 个包，确认一次。或者是收到了乱序的数据包会立马确认。

也就是收到前几个数据包先不确认，到第几个数据包回复一个ACK一起确认。这样能减少网络中数据包的个数，这样的方式就称为延时确认。

在许多情况下，TCP并不对每个到来的数据包都返回ACK，利用TCP的累积ACK字段就能实现该功能。累积确认可以允许TCP延迟一段时间发送ACK，以便将ACK和相同方向上需要传的数据结合发送。这种捎带传输的方法经常用于批量数据传输。显然，TCP不能任意时长地延迟ACK;否则对方会误认为数据丢失而出现不必要的重传。

3.9 拥塞控制

要考虑的问题：

1、什么是网络拥塞

2、怎么判断网络出现拥塞

3、一旦出现网络拥塞，我们应该怎么做

1、什么是网络拥塞

就像一条公路上出现事故，或者在施工，车特别多，但是所有车都往前挤，挤到最后就堵住了。

相比于网络中，数据的转发要经过路由器，路由器转发数据也不是瞬时完成的，他需要进行解析，才能确定应该转发到哪个路由器。但是一旦网络中是数据包很多很多，路由器一下子是处理不过来的，只能放到路由器的缓冲区中排队等待处理（这个也就是导致网络延迟的一部分原因），但是这样好歹数据包是能发送到的，只不过是延时高一些。

但是，如果数据包特别多，导致路由器的缓冲区满了，此时就产生了网络拥塞现象，此时再向路由器发送数据包之后，就会造成数据包丢失（造成丢包），而我们使用的是TCP协议，丢包之后有重传机制，这个时候路由器越满---》丢包越多---》重传越多---》路由器缓冲区越满，就形成了恶性循环。是大多数客户端使用网络造成的影响。

2、怎么判断网络出现拥塞

就和堵车一样，你怎么知道前面堵不堵车，一是交警告诉你前面堵车了，二是你堵住了就知道堵车了

相比于网络，一是由 路由器 通知终端，给终端发送一个ICMP的报文，告诉终端我拥塞了，你别发了，这样是最直接最方便的方式，但是，路由器都满了，你还让他往外发，更增加负担了。

二是由终端自己判断，当客户端发生了超时重传的时候，他就可以认为是当前路由器出现拥塞了，超时重传的大概率原因是丢包，而丢包的大概率原因是因为网络拥塞。

3、怎么解决网络拥塞

前面我们说了，网络拥塞现象的本质其实是大多数客户端使用网络造成的，而路由器的性能是固定的，你没办法去立马改变路由器性能，我们应该解决根本的问题，也就是要减少终端向网络中传输数据的速度（也就是慢点发或者是别发，那都堵住了，你还发），此时路由器就会处理缓冲区中的数据包，等路由器处理完之后，再发

也就是一旦判断为网络拥塞，发送方就会降低终端向网络中注入数据的速度。

一旦降低速度，那么也就引出下面几个问题

（1）如何控制向网络中注入数据的速度？

（2）速度该降低到多少？要降低到什么时候？什么时候恢复为正常速度？怎么恢复到正常速度？

（1）如何控制向网络中注入数据的速度？

由一个变量控制，CWND（拥塞窗口的大小），AWND（通告窗口大小）

流量控制是由滑动窗口机制管理的，而窗口大小 = min(AWND,CWND)

（2）速度该降低到多少？要降低到什么时候？什么时候恢复为正常速度？怎么恢复到正常速度？

本质就是 发送方该如何控制CWND的变化，也就是改变窗口大小

慢启动算法

拥塞避免算法

快重传

快恢复

改变窗口大小的算法

慢启动算法：每经过一个RTT（传输轮次，一个数据包发送到应答），CWND*2，呈现指数级增长，但也不是无限的，他存在一个阈值h，一开始是使用慢启动算法，一旦到达这个阈值，就只能使用拥塞避免算法。

拥塞避免算法：每经过一个RTT，CWND+1,线性增长

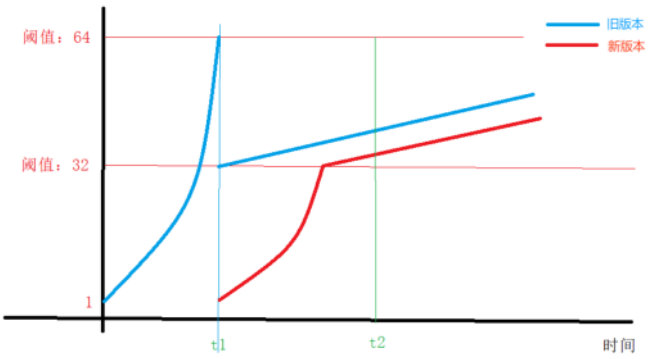
• 而慢启动为什么叫慢启动？他不是按照指数级增长么？

什么时候执行慢启动算法？

- CWND < 阈值
- 慢启动算法的早期版本：一旦发生网络拥塞，慢启动阈值变为原来窗口大小的一半，阈值= CWND/2，然后执行拥塞避免算法。
- 慢启动算法的最新版本：一旦发生网络拥塞，慢启动阈值变为原来窗口大小的一半，CWND=1，继续执行慢启动算法缓冲算法：阈值= CWND/2,CWND=1,然后呈指数增加

什么时候执行拥塞避免算法？（线性增加）

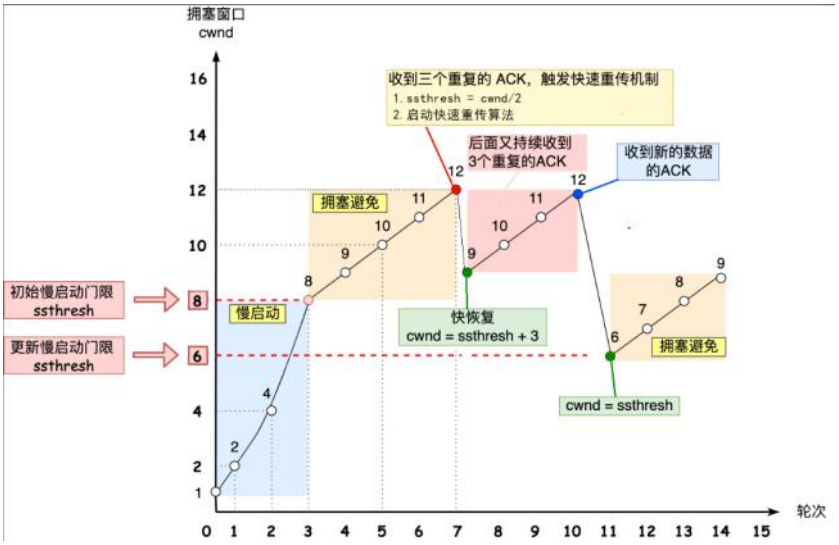
- CWND > 阈值



如上图是 慢启动到达阈值之后，直接使用**旧版本** 和 使用 **新版本**的区别，在网络拥塞之后，慢启动能够极大的降低窗口大小（也就是传输速度），同时又能快速增加窗口大小到新的阈值，这样相比于旧版本，新版本在任意时间上的发送速度都会小于旧版本。在发生拥塞的时间段内，会大大降低发生速度。

快重传：快重传，数据包的丢失不是因为拥塞控制，因此不用直接使用慢启动，而是启动快速重传机制，此时将阈值等于窗口大小的一半，阈值 = CWND/2,

快恢复：而是将窗口大小调整为阈值+3，CWND = 阈值+3，
然后每收到一个重复的ACK,CWND就接着加一（快恢复过程），直到收到新的ACK确认，将当前窗口大小修改为阈值，CWND= 阈值，然后执行拥塞避免算法。


拥塞窗口
cwnd
收到三个重复的 ACK，触发快速重传机制
1. ssthresh = cwnd/2
2. 启动快速重传算法
后面又持续收到
3个重复的ACK
收到新的数据
的ACK
拥塞避免
初始慢启动门限
ssthresh
更新慢启动门限
ssthresh
慢启动
快恢复
cwnd = ssthresh + 3
拥塞避免
cwnd = ssthresh
轮次